AFRL-IF-RS-TR-2002-1
Final Technical Report
January 2002

EXOKERNEL AND EXOTOOLS

Massachusetts Institute of Technology

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F270

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-1 has been reviewed and is approved for publication.

APPROVED: Kevin A. Kwiat

KEVIN A. KWIAT
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	JANUARY 2002	Final Jun 97 - Dec 00

4. TITLE AND SUBTITLE
EXOKERNEL AND EXOTOOLS

5. FUNDING NUMBERS
C - F30602-97-2-0288
PE - 62301E
PR - F270
TA - 71
WU - 04

6. AUTHOR(S)
M. Frans Kaashoek, Charles Blake, and Robert Morris

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

Air Force Research Laboratory/IFGA
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2002-1

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Kevin A. Kwiat/IFGA/(315) 330-1692

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The Internet has brought a qualitative change in operating system (OS) requirements. Firewalls will not be able to cope with exploding application demand for seamless network access: OS support for security is needed. Similarly, OS support for fast data service is vital when most applications serve data. Today's operating systems do not support these requirements. This report describes technologies for high-performance Internet computing based on exokernel architectures. The exokernel is extremely flexible as it implements conventional OS abstractions as application libraries, enabling very fast innovation at all system levels. For instance, enhancements to network protocols can be distributed with the applications that require them, rather than years later in the next OS release.

14. SUBJECT TERMS
Exokernel, Exotools, Operating System, OS

15. NUMBER OF PAGES
32

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

1 Overview

This report is the final report for the Exokernel Operating System project sponsored by DARPA under the Quorum program. The Internet has brought a qualitative change in operating system (OS) requirements. Firewalls will not be able to cope with exploding application demand for seamless network access: OS support for security is needed. Similarly, OS support for fast data service is vital when most applications serve data. Today's operating systems do not support these requirements. This report summarizes the operating system architecture that we have designed and the prototype that we have built that meets these requirements.

Our technologies for high-performance Internet computing are based on the *exokernel architecture*, a new operating system developed with DARPA funding. The exokernel is extremely flexible. It implements conventional operating system abstractions as application libraries, enabling very fast innovation at all system levels. For instance, enhancements to network protocols can be distributed with the applications that require them, rather than years later in the next OS release.

In June 1997 DARPA funded us to develop two new technologies on top of the exokernel base.

- **Exokernel security.** Conventional operating systems have serious security problems: processes run with too much privilege, and inadequate system call interfaces make applications prone to vulnerabilities. To address these problems, we proposed a set of exokernel security techniques based on a uniform discretionary access control mechanism (*hierarchically-named capabilities*). We built a new exokernel based on these techniques. This exokernel simplified the construction of secure applications, greatly reducing the need to rely on complex firewalls for security.

- **Tools for high-performance, secure, and correct servers.** The abstractions provided by current operating systems often make servers *harder* to write and limit their performance through software overheads and inefficient resource use. This is unacceptable for the Internet environment, where most applications have some server functionality. We propose new tools and abstractions to build high-performance, correct, and secure servers.

In this report we summarize the development of the these new technologies, as well as a number of new technologies (such as Click) that we developed as a result of the work under this contract.

- Xok, an exokernel operating system with support for hierarchically-named capabilities. We designed and built this as originally proposed.

- The Cheetah high-performance web server. This server was not proposed originally, but became an additional result of the research.

- The Prolac protocol compiler. We designed and built the language, compiler, and TCP networking stack as originally proposed.

- The 'C language and compiler for creating efficient dynamic code. We designed and built the language and compiler, and demonstrated its effectiveness on a number of applications.

- The Click modular software router. We designed and built the Click modular router to facilitate the easy construction of applications that run inside the network (such as firewalls). Click was not proposed originally, but became an additional result of the research.

Cheetah and Click were not proposed under the project, but grew out of the research proposed. Interestingly, Cheetah and Click are the deliverables that are having the highest practical impact.

The rest of this report is structured around these deliverables (Section 2). The report concludes with comments on technology transfer (Section 3) and a list of publications (Section 4).

Much of this information is also available on our Web site, `http://pdos.lcs.mit.edu`. This report borrows excerpts from publications that were sponsored under this grant.

2 Results

This section motivates the research projects that were funded under this grant and then describes the results we obtained. Section 2.1 describes the exokernel operating system architecture and the high-performance Cheetah Web server that exploits the advantages that the exokernel architecture offers. Section 2.2 describes the Prolac protocol compiler, which makes it easy to develop protocol stacks for applications such as Cheetah. The initial exokernel exploited dynamic code generation in an ad-hoc fashion—Section 2.3 presents 'C, a general tool for efficient, portable dynamic code generation. Finally,Se ction 2.4 introduces Click, a modular software router toolkit, which allows easy development of applications that run in the network.

2.1 The Exokernel

In traditional operating systems, only privileged servers and the kernel can manage system resources. Untrusted applications are restricted to the interfaces and implementations of this privileged software. This organization is flawed because application demands vary widely. An interface designed to accommodate every application must anticipate all possible needs. The implementation of such an interface would need to resolve all tradeoffs and anticipate all ways the interface could be used. Experience suggests that such anticipation is infeasible and that the cost of mistakes is high [2, 3, 9, 13, 15, 32].

The *exokernel architecture* [13] solves this problem by giving untrusted applications as much control over resources as possible. It does so by dividing responsibilities differently from the way conventional systems do. Exokernels separate protection from management: they protect resources but delegate management to applications. For example, each application manages its own disk-block cache, but the exokernel allows cached pages to be shared securely across all applications. Thus, the exokernel protects pages and disk blocks, but applications manage them.

Of course, not all applications need customized resource management. Instead of communicating with the exokernel directly, we expect most programs to be linked with libraries that hide low-level resources behind traditional operating system abstractions. However, unlike traditional implementations of these abstractions, library implementations are unprivileged and can therefore be modified or replaced at will. We refer to these unprivileged libraries as *library operating systems,* or libOSes.

We hope the exokernel organization will facilitate operating system innovation: there are several orders of magnitude more application programmers than OS implementors, and any programmer can specialize a libOS without affecting the rest of the system. LibOSes also allow incremental, selective adoption of new OS features: applications link with the libOSes that provide what

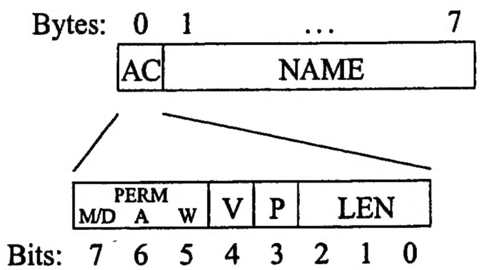

Figure 1: Structure of hierarchically-named capability

they need—new OS functionality is effectively distributed with the application binary.

2.1.1 Hierarchical Capabilities

A lack of flexibility in today's multi-user operating systems seriously hurts system security. The principle of least privilege states that a process should have access to the smallest number of objects necessary to accomplish a given task. At the level of system calls, conventional operating systems provide only two privilege levels: "root" and "some user." The lack of privilege definition flexibility forces designers to allow a large selection of complex daemons to execute as root. The excessive privilege granted to these applications greatly enlarges the amount of code to which one must entrust system security.

Moreover, since these trusted programs go through the system call interface, it becomes easy to fall victim to a large class of security bugs related to short periods between the time-of-check and the time-of-use. Seemingly simple tasks as opening a file only if it belongs to a particular user become very tricky.

While people have long complained about the need to run a great deal of software as root, equal permission to every file of any given user is also problematic. Many programs require only read permission on input files and write permission on output files, but Unix does not allow easy ways to express such relationships accurately. This allows trojan horse and rogue programs to do much more damage.

The exokernel system introduces hierarchically named capabilities as a simple, flexible access control mechanism. A capability is just a string of opaque bytes with a header. This string describes the credentials needed to access hardware objects such as memory pages or disk blocks. The semantics of the name specified by the bytes are decided by the libOS.

More concretely, figure 1 shows the layout of a hierarchically-named capability. A capability is 8 bytes long. The first byte header records the properties of a capability and the other 7 bytes its name. The properties in the first byte include the length of the name (from 0 to 7 bytes), a valid bit, a pointer bit (for extended ACLs), and three permissions bits: modify ACL/duplicate capability, allocate resources, and write.

The kernel maintains a list of the capabilities owned by each running task. Every system call that accesses guarded resources first checks that the calling task owns an adequate capability. Hierarchy is generated by considering an owned capability adequate if it is a *prefix* of the guarding capability. In this case we say that the shorter capability *dominates* the longer.

Capabilities are granted to new processes by their creators, e.g. a login process or shells. The

hierarchical nature of the naming allows a creator process to carefully refine the privileges passed on to new processes.

The exokernel capability system demonstrates that a flexible operating system security model can simplify the construction of secure applications and decrease the likelihood of security implementation mistakes. Two different companies chose to use the exokernel specifically for its security model—one to sandbox downloaded code in active-network nodes, the other to flexibly restrict the privilege of server-side web scripts.

2.1.2 Protected Methods

The exokernel decomposition of system software puts each program in control of how hardware is abstracted, but some OS abstractions operate across programs. Protected methods and the protected abstraction mechanism enable such multi- program abstractions to be built with fine-grained control over how much the two programs must trust each other.

Consider a pair of programs sharing a physical memory page as a data "pipe". Programs do not trust each other to manipulate this buffer in arbitrary ways. In particular, the reader wants to prevent the writer from overwriting data that the reader has not yet accessed (or copied into reader-controlled memory). Hardware memory management allows the writer to write anywhere in the page at any time. Additional policy is required to allow and revoke writes by the writer in concert with the reader draining the buffer.

The exokernel philosophy is to provide mechanism, but not policy. One answer along these lines is that buffer updates be made through kernel calls. The kernel's priviledged address space holds the state used to enforce writing only when there is available head-of-buffer space. In addition to the overhead involved, this fixes the policies implemented by the kernel, an effect the exokernel tries to avoid if possible. Another answer is the protected method and protected abstraction mechanisms (PAM) that we devised.

PAM guards an abstraction with a capability. When a process invokes the method the kernel endows it with that capability only for the duration of the function call. Invoking programs are required only to trust that priviledge process that installs the protected abstraction.

PAM uses the hierarchical capabilities to allow unpriviledged, untrusted applications to define and securely share generic, stateful abstractions at run-time. PAM achieves a good flexibility-performance combination by eliminating the need for context switches and optimizing for the common case, in which the same abstraction is invoked repeatedly. PAM's design emphasizes simplicity and correctness, which makes it easy to understand and use.

2.1.3 Virtual Memory

Unlike the MIPS architecture studied in previous exokernel research, the x86 architecture defines a virtual page-table structure. The MMU hardware itself handles refilling the virtual-to-physical translation cache. We were able to give applications the ability to control other aspects of virtual memory, such as read, write, or execute access rights, on a per-page basis. Additionally, x86 hardware allows three system software-defined bits to be used. In a libOS we use one of these bits to decide whether a virtual page is resident in memory or needs to be fetched from the disk.

MMU exceptions are delivered directly to processes (or libOS's) by an upcall. This upcall gets the offending address. If it corresponds to a non-resident page, the libOS will read the page

from disk into a physical page and then make the kernel call to bind the page into the page table. If the address is truly invalid the user-level MMU exception handler can decide whether to abort execution, dump a memory image to disk, or somehow resume execution.

We have found application-level virtual memory (AVM) to have many advantages. AVM adeptly delegates the issue of page replacement to the user-level process. The program itself is the entity most able to rank the priority of its memory pages. Since the x86 and many other CPUs have physically mapped caches, precise control over how physical pages are laid out in memory gives a program better control over its cache usage. Knowing what pages are and are not resident in the context of a program execution may be of dramatic advantage for programs with large working sets that are accessed in a predictable way. For example, scientific programs can asynchronously pre-fetch all the pages they are going to need well in advance of when they need them.[25] Some systems, such as x86, allow different sized memory pages, but it would be difficult for a general purpose kernel to anticipate what the right size is for a given application. Some garbage collection strategies can be made much more efficient if they are allowed access to the MMU's unreserved bits to store page accounting data.

2.1.4 Disk Subsystems: XN

One goal of a secure, flexibile operating system architecture is that users can define the right file system for their application, without compromising the security of other applications. Determining the layout of a file system is, important, for example for the Cheetah Web server. Designing a flexible stable storage system has proven difficult: XN is our fourth design. This section provides an overview of UDFs, the cornerstone of the final XN.

XN provides access to stable storage at the level of disk blocks, exporting a buffer cache registry as well as free maps and other on-disk structures. The main purpose of XN is to determine the access rights of a given principal to a given disk block as efficiently as possible. XN must prevent a malicious user from claiming another user's disk blocks as part of his own files. On a conventional OS, this task is easy, since the kernel itself knows the file's metadata format. On an exokernel, where files have application-defined metadata layouts, the task is more difficult.

XN's novel solution employs *UDFs* (*untrusted deterministic functions*). UDFs are metadata translation functions specific to each file type. XN uses UDFs to analyze metadata and translate it into a simple form the kernel understands. A libFS developer can install UDFs to introduce new on-disk metadata formats. The restricted language in which UDFs are specified ensures that they are deterministic—their output depends only on their input (the metadata itself). UDFs allow the kernel to safely and efficiently handle any metadata layout without understanding the layout itself.

UDFs are stored on disk in structures called *templates*. Each template corresponds to a particular metadata format; for example, a UNIX file system would have templates for data blocks, inode blocks, inodes, indirect blocks, etc. Each template T has one UDF: *owns-udf*$_T$,a nd two untrusted but potentially nondeterministic functions: *acl-uf*$_T$ and *size-uf*$_T$. All three functions are specified in the same language but only *owns-udf*$_T$ must be deterministic. The other two can have access to, for example, the time of day. The limited language used to write these functions is a pseudo-RISC assembly language, checked by the kernel to ensure determinacy. Once a template is specified, it cannot be changed.

For a piece of metadata m of template type T, *owns-udf*$_T(m)$ returns the set of blocks which

m points to and their respective template types. UDF determinism guarantees that *owns-udf* will always compute the same output for a given input: XN cannot be spoofed by *owns-udf*. The set of blocks *owns-udf* returns is represented as a set of tuples. Each tuple constitutes a range: a block address that specifies the start of the range, the number of blocks in the range, and the template identifier for the blocks in the range.

Once installed, types are persistent across reboots. To ensure that libFS data is persistent across reboots, a libFS can register the root of its tree in XN's *root catalogue*. A root entry consists of a disk extent and corresponding template type, identified by a unique string (e.g., "mylibFS"). After a crash, XN uses these roots to garbage-collect the disk by reconstructing the free map. It does so by logically traversing all roots and all blocks reachable from them: reachable blocks are allocated, non-reachable blocks are not.

Finally, there is the XN buffer cache registry, which allows protected sharing of disk blocks among libFSes. The registry tracks the mapping of cached disk blocks and their metadata to physical pages (and vice versa). Unlike traditional buffer caches, it only records the mapping, not the disk blocks themselves. The disk blocks are stored in application-managed physical-memory pages. The registry tracks both the mapping and its state (dirty, out of core, uninitialized, locked). To allow libFSes to see which disk blocks are cached, the buffer cache registry is mapped read-only into application space. Registry entries can be inserted without requiring that the object they describe be in memory. Blocks can also be installed in the registry before their template or parent is known.

XN does not replace physical pages from the registry (except for those freed by applications), thus allowing applications to determine the most appropriate caching policy. Because applications also manage virtual memory paging, the partitioning of disk cache and virtual memory backing store is under application control.

To allow the construction of daemons that asynchronously write dirty blocks, XN allows any process to write out blocks not associated with any running process, even if that process does not have specific permission. LibFSes do not have to trust daemons with write access to their files, only to flush the blocks. This ability has three benefits. First, the contents of the registry can be safely retained across process invocations rather than having to be brought in and paged out on creation and exit. Second, this design simplifies the implementations of libFSes, since a libFS can rely on a daemon of its choice to flush dirty blocks even in difficult situations (e.g., if the application containing the libFS is swapped out). Third, this design allows different write-back policies.

In summary, our experience with XN shows that, even with the strongest desired guarantees, a protected interface can still provide significant flexibility to unprivileged software. We also found that the exokernel approach can deal as readily with high-level protection requirements as it can with those closer to hardware.

2.1.5 Multiprocessor

The original design of the exokernel system did not support PC multiprocessor architectures. We developed a symmetric multiprocessing exokernel and demonstrated that unprivileged library implementation of operating system abstractions is viable on a multiprocessor system.

Our research focused on three issues. First, it studied synchronization options for the kernel. Second, we developed three new exokernel interfaces: message passing, kernel support for

multithreading, and multiprocessor scheduling. Third, because exokernel applications do not trust each other, traditional synchronization primitives used to guard system abstractions, such as voluntary memory locks and semaphores, do not satisfy our needs. A multiprocessor exokernel and a synchronized library operating system resulted from this research.

Our research evaluated a strategy for synchronization among untrusted processes we call copy-modify-replace (CMR). First a writer process makes a private copy of the shared data, but makes it readable by others. Then the private copy is modified. Finally, if no contention has occurred, an atomic pointer switch establishes the new copy as the current copy. Contention is found by either correctness checks on the shared data or by detecting the violation of a voluntary mutual exclusion protocol. If contention is found then the critical section is either re-tried or finally terminated. The CMR algorithm is optimistic in that it is fast when access to shared state does not collide repeatedly. A malicious process can significantly degrade CMR performance by engineering such collisions.

Performance analysis showed that the overheads of synchronization in both the kernel and the library operating system are small. For example, we found total synchronization overhead to be less than 25% of total operation cost for common OS operations like creating new processes. Of this overhead, less than 2% was overhead due to CMR-induced costs.

2.1.6 Dynamic Packet Filter Improvements

Exokernels do not have traditional network stacks. Instead, dynamic packet filters (DPFs) run during network interrupts to decide which user-level packet rings incoming packets should be copied to. Later, when processes run in their own scheduling contexts, they interpret their packet rings to implement full protocols such as TCP. Packets that do not match any filters are dropped. Each filter is a sequence of one or more atoms declaring facts about various offsets within the packet.

To support a variety of links on Internets, the IP protocol provides for packet fragmentation, requiring end hosts to implement re-assembly. Except for the zero-offset fragment, IP fragments contain only the packet ID and the offset of the fragment within the larger packet. Higher level protocol headers exist only in received packets corresponding to the zero-offset fragment. The intervening network may re-order packets, allowing fragments to arrive in any order. Thus the decision of which buffer to store packets in cannot occur prior to reception of the zero-offset fragment. Reassembly requires matching chains of packets with the same IP ID. Besides network re-ordering, sending operating systems, such as Linux, may even opt to specifically transmit fragments in reverse order.

Supporting UDP packet fragmentation alerted us to inadequacies in the original DPF design and an issue with the original privilege analysis.

The original filter design could could not handle stateful packet chains in which a filter pattern is parametrized by the content of earlier packets (specified by some other filter, of course). We solved this problem by adding two new atoms to the filter mechanism. The first atom allows a restricted action to occur. In the case of fragments it can store the IP identifier. The second new atom can be parameterized by this state to match fragments from this chain. This effectively lets the filter change rapidly with each new received packet. There may be non-IP-stack related applications of this sort of stateful coordination of filters. Our performance tests indicated that this

is effective, with our fragment processing chain within taking only on the order of 25% more time than a conventional monolithic kernel IP stack.

A thorny privilege issue was also raised, however. Delegating memory allocation to user-level programs assumes that all packets can be classified immediately (i.e., inside the network interrupt handler). While this is still true in the common case, it is not true for out-of-order fragments whose zero-offset fragment arrives late or never arrives at all. If a packet with a new IP identifier arrives, and it is not the zero-offset fragment, the packet must be buffered, but there is no natural user-level place to store it. Therefore it must be buffered either by the kernel or by a process which can add to any packet ring. Since the zero-offset packet for this IP identifier may never be received, the privileged memory must also be time-limited and space-limited. We have preferred to keep this sort of policy-heavy management out of Xok proper. The appropriate design would seem to be a user-level server dedicated to managing memory for unmatched or not yet matched UDP fragments. Applications receiving fragmented UDP packets would give this server the capabilities to add packets to the appropriate packet rings.

2.1.7 Drivers

Any new PC operating system faces the hurdle of supporting a vast variety of hardware, even just to get others to experiment with it. The possibility of porting drivers from other open source operating systems exists, but even this often requires substantial effort. Part of our ongoing work with the exokernel was to facilitate other groups using it as a basis for experimentation. Two kinds of device drivers proved particularly useful. First, the exokernel's ability to boot from a network filesystem meant that support for network interfaces was more important than for disks. Second, for applications requiring disks, the pervasiveness of IDE disks compared to SCSI disks made support for the former the most important.

OSKit Network drivers The University of Utah Fluke project developed a slow but functional technique to access Linux and FreeBSD device drivers through bridge interfaces. Their OSKit exports a single network device driver that can bind to many common underlying network interface cards. We ported just this single exported driver to the exokernel. This allowed access for other researchers with a variety of common network cards, including, for example, the 3COM and Intel Ethernet Express cards.

UltraDMA IDE disk driver IDE originally required a fairly high overhead per bus transaction. The current generation of IDE controllers added better support for direct memory access data transfers. That type of support was very close to the assumptions the exokernel disk system had been making about SCSI disks. We ported the FreeBSD UltraDMA driver to the exokernel. We also had to improve support for various aspects of the disk system, such as partition tables and sector transfer limits. We updated the driver as the FreeBSD driver was improved to add support for more common motherboard chip sets (Intel PIIX 3, 4, etc., Via). We tested the driver with several drives and performance and hardware compatibility was comparable to FreeBSD.

2.1.8 Linux Technology Transfer

We have also explored how the advantages of an exokernel can be brought to conventional operating systems in an evolutionary way. Our focus has been on disk and network subsystems since device management seems to matter most in modern high-end applications such as web servers. We integrated two exokernel subsystems into Linux—XN for low level disk access and DPF for managing raw network access. This demonstrated that we could make low-level control available to applications without forcing applications to abandon all other Unix abstractions.

Integrating DPF into the packet processing chain was straightforward and resulted in a 30-100% increase in the end-to-end performance of the Cheetah-DPF web server over Harvest.

Integrating XN into Linux was more difficult and performance harder to quantify in an end-to-end comparison. Microbenchmarks indicated that primitive disk operations through XN were actually slightly faster than the corresponding Linux interfaces to raw disk devices. This pathway in Linux is not what filesystems use and is consequently less optimized. Unlike its network stacks, internal Linux interfaces for the buffer cache and disk have not been designed with the idea of interposing arbitrary processing.

We also added the exokernel's generic batched system calls to Linux. Generic batching allows interfaces to remain low-level and hardware less abstracted while preventing an expensive explosion in number of user-kernel boundary crossings.

One of our conclusions from this effort was that exokernel mechanisms can be added to conventional operating systems fairly easily; the challenges mostly involve understanding the operating system, rather than the exokernel implementation.

2.1.9 Cheetah

The exokernel architecture is well suited to building fast servers (e.g., NFS servers or web servers). Server performance is crucial to client/server applications [16], and the I/O-centric nature of servers makes operating system-based optimizations profitable.

We have developed an extensible I/O library (XIO) for fast servers and a sample application that uses it, the Cheetah HTTP server. This library is designed to allow application writers to exploit domain-specific knowledge and to simplify the construction of high-performance servers by removing the need to "trick" the operating system into doing what the application requires (e.g., Harvest [7] stores cached pages in multiple directories to achieve fast name lookup).

An HTTP server's task is simple: given a client request, it finds the appropriate document and sends it. The Cheetah Web server performs the following set of optimizations as well as others not listed here. These optimizations performed by Cheetah are architecture independent.

- **Merged File Cache and Retransmission Pool.** Cheetah avoids all in-memory data touching (by the CPU) and the need for a distinct TCP retransmission pool by transmitting file data directly from the file cache using precomputed file checksums (which are stored with each file). Data are transmitted (and retransmitted, if necessary) to the client directly from the file cache without CPU copy operations. (Pai et al. have also used this technique [26].)

- **Knowledge-based Packet Merging.** Cheetah exploits knowledge of its per-request state transitions to reduce the number of I/O actions it initiates. For example, it avoids sending

redundant control packets by delaying ACKs for client HTTP requests, since it knows it will be able to piggy-back them on the response. This optimization is particularly valuable for small document sizes, where the reduction represents a substantial fraction (e.g., 20%) of the total number of packets.

- **HTML-based File Grouping**. Cheetah co-locates files included in an HTML document by allocating them in disk blocks adjacent to that file when possible. When the file cache does not capture the majority of client requests, this extension can improve HTTP throughput by up to a factor of two.

Our base HTTP server performs roughly as well as the Harvest cache, which has been shown to outperform many other HTTP server implementations on general-purpose operating systems. This gives us a reasonable starting point for evaluating extensions that improve performance.

The default socket and file system implementations built on top of XIO perform significantly better than the OpenBSD implementations of the same interfaces (by 80-100%). The improvement comes mainly from simple (though generally valuable) extensions, such as packet merging, application-level caching of pointers to file cache blocks, and protocol control block reuse.

Most notably, Cheetah significantly outperforms the servers that use traditional interfaces. By exploiting Xok's extensibility, Cheetah gains a four times performance improvement for small documents (1 KByte and smaller), making it eight times faster than the best performance we could achieve on OpenBSD. Furthermore, the large document performance for Cheetah is limited by the available network bandwidth (three 100Mbit/s Ethernets) rather than by the server hardware. While the socket-based implementation is limited to only 16.5 MByte/s with 100% CPU utilization, Cheetah delivers over 29.3 MByte/s with the CPU idle over 30% of the time. The extensibility of ExOS's default unprivileged TCP/IP and file system implementations made it possible to achieve these performance improvements incrementally and with low complexity.

2.1.10 Project history

The exokernel operating system architecture was developed under a previous DARPA contract. The approach looked promising, but also raised many questions. Can ambitious applications actually achieve significant performance improvements on an exokernel? Will traditional applications—for example, unaltered UNIX applications—pay a price in reduced performance? Is global performance compromised when no centralized authority decides scheduling and multiplexing policies? Does the lack of a centralized management policy for shared OS structures lower the integrity of the system?

The research funded under this contract attempts to answer these questions and thereby evaluate the soundness of the exokernel approach. Our experiments are performed on the Xok/ExOS exokernel system. Xok is an exokernel for Intel x86-based computers and ExOS is its default libOS (as described above in the previous sections). Xok/ExOS compiles on itself and runs many unmodified UNIX programs (e.g., perl, gcc, telnet, and most file utilities). We compare Xok/ExOS to two widely-used 4.4BSD UNIX systems running on the same hardware, using large, real-world applications.

ExOS ensures the integrity of many of its abstractions using Xok's support for protected sharing. Some abstractions, however, still use shared global data structures. ExOS cannot guarantee

UNIX semantics for these abstractions until they are protected from arbitrary writes by other processes. In our measurements, we approximate the cost of this protection by inserting system calls before all writes to shared global state.

Our results show that most unmodified UNIX applications perform comparably on Xok/ExOS and on FreeBSD or OpenBSD [17]. Some applications, however, run up to a factor of four faster on Xok/ExOS. Experiments with multiple applications running concurrently also show that exokernels can offer competitive global system performance.

We have also demonstrated that application-level control can significantly improve the performance of applications. For example, we describe a new high-performance HTTP server,C heetah, that actively exploits exokernel extensibility.C heetah uses a file system and a TCP implementation customized for the properties of HTTP traffic. Cheetah performs up to eight times faster than the best UNIX HTTP server we measured on the same hardware.

The positive results stimulated much acadamic and commercial interest. Many exokernel ideas were adopted by other systems. The exokernel publications are among the most cited papers in the operating systems community.

During 1998 and 1999, the software distribution attracted an active user community. One other research project, at Trusted Information Systems (TIS), sponsored by DARPA (under the active networks program) used the software we developed. In late 1998 the key development team (master's and junior Ph.D. students) at MIT left MIT to found a company based on the exokernel technology. The company sells cost-effective high-performance video servers (see http://www.vividon.com). After the core development team left, our efforts focused on other aspects of the research, as described in the next sections.

2.2 Prolac

As part of developing library operating systems for ExOS, we found ourselves spending much time extending the TCP protocol stack. This task turned out to be a difficult, and we therefore decided to develop a tool for making the development of protocol stacks easier. The result of that research is the Prolac protocol compiler.

2.2.1 Prolac

Designing and implementing network protocols with conventional languages and tools is difficult. The protocols themselves are hard to design; implementing a protocol correctly is another challenge [35]. Furthermore, protocol efficiency has become vital with the growing importance of networking, and the occasional need for protocol extensions [34, 5] only complicates the issue. Unfortunately, these tensions work against one another. Many optimizations which make protocol code more efficient also tend to make it much harder to understand [24], and therefore harder to get right. Extensions affect deeply buried snippets of protocol code rarely identifiable a priori. Finally, the clearest organization of protocol code is often among the slowest.

Specialized language tools are a natural area to investigate for a solution to this software engineering problem. Most previous work, however, has focused on only one of the three issues: correctness. Research has been particularly active in formal specification languages amenable to machine verification [4, 10]; while it is possible to use one of these specification languages to

generate an implementation semi-automatically, very high performance is often precluded by the languages themselves. Some work has focused on high-performance implementation [1, 6], but these languages may not be suitable for existing protocols, either by design or due to limitations in the underlying language model.

The Prolac language, and its compiler, *prolacc*, address all three issues in protocol implementation: correctness, efficiency, and extensions. Our protocol compiler project has three specific goals: to implement protocol-specific optimizations, thus creating high-performance protocol implementations; to facilitate protocol extensions; and to make protocol implementations more tractable to human readers, and thus easier for people to reason about.

Prolac is a statically-typed, object-oriented language. Unlike many such languages, it focuses on small functions rather than large ones: its syntax encourages the programmer to divide computation into small pieces, and Prolac features, such as namespaces, help a programmer name such small pieces appropriately. A protocol specification divided into small rules is both easier to read and easier to change; a small extension is more likely to affect just a few small functions in Prolac, which can be overridden by a subtype without complicating the base protocol code. Several novel features—specifically, module operators and implicit rules—help make protocol specifications easier to understand.

The *prolacc* compiler compiles a Prolac specification to C code. It applies several protocol-specific optimizations, such as extensive inlining and outlining; these optimizations are specified in the Prolac language as annotations to modules or to individual expressions. Both the language and the compiler have been designed to produce efficient generated code—our goal was to equal or exceed the performance of protocol implementations hand-written in C. In particular, *prolacc* can analyze away almost every dynamic dispatch and structure assignment.

We have designed and developed a prototype implementation of the TCP protocol [31] in the Prolac language, written largely as a proof of concept. We focused on TCP because it is a large, complex, important, and well-documented protocol. TCP is widely recognized as being difficult to implement well; in fact, books have been written about its implementation [35].

Our Prolac TCP specification is divided into small, sensibly interlocked pieces. Logical extensions to the base TCP protocol, such as delayed acknowledgement and slow start, are implemented in the specification as extensions to a set of base modules; very simple definitions determine which extensions, if any, are compiled, and even their relative order of execution. The TCP specification is highly extensible while staying highly readable—much of the specification is very similar to language in the standard reference to TCP [31]. Prolac TCP can communicate with other TCPs, and experiments show that Prolac is not a bottleneck under normal networking conditions.

2.2.2 Project history

Prolac was essentially a small project with a single person (Eddie Kohler) driving it. It was intellectually successful (we showed it could be done), and resulted in a major publication, which appeared at SIGCOMM [20]. In practice, however, it had little impact—a few projects used the public distribution of the software, but in general people weren't willing learn a new programming language. Because of this experience, the Click project (Section 2.4) used a standard computer language, and is having much bigger practical impact.

2.3 Tcc

The exokernel operating system uses dynamic code generation for efficiency (e.g., to compile packet filters when they are installed). As support for such usages of dynamic code generation, we wanted a tool that would allow us to easily write portable, efficient dynamic code. The result of that research is 'C (pronounced as tick-C).

2.3.1 'C

Dynamic code generation—the generation of executable code at *run time*—enables the use of run-time information to improve code quality. Information about run-time invariants provides new opportunities for classical optimizations such as strength reduction, dead-code elimination, and inlining. In addition, dynamic code generation is the key technology behind just-in-time compilers, compiling interpreters, and other components of modern mobile code and other adaptive systems.

'C is a superset of ANSI C that supports the high-level and efficient use of dynamic code generation. It extends ANSI C with a small number of constructs that allow the programmer to express dynamic code at the level of C expressions and statements, and to *compose* arbitrary dynamic code at run time. These features enable programmers to write complex imperative code manipulation programs in a style similar to LISP [33], and they make it relatively easy to write powerful and portable dynamic code. Furthermore, since 'C is a superset of ANSI C, it is not difficult to improve performance of code incrementally by adding dynamic code generation to existing C programs.

'C's extensions to C—two type constructors, three unary operators, and a few special forms—allow dynamic code to be type-checked statically. Much of the overhead of dynamic compilation can therefore be incurred statically, which improves the efficiency of dynamic compilation. While these constructs were designed for ANSI C, it should be straightforward to add analogous constructs to other statically typed languages.

tcc is an efficient and freely available implementation of 'C, consisting of a front end, back ends that compile to C and to MIPS and SPARC assembly,a nd two run-time systems. tcc allows the user to trade dynamic code quality for dynamic code generation speed. If compilation speed must be maximized, dynamic code generation and register allocation can be performed in one pass; if code quality is most important, the system can construct and optimize an intermediate representation prior to code generation. The overhead of dynamic code generation is approximately 100 cycles per generated instruction when tcc only performs simple dynamic code optimization, and approximately 600 cycles per generated instruction when all of tcc's dynamic optimizations are turned on.

The research makes the following contributions:

- The 'C language.
- tcc, the 'C compiler, in particular, its two run-time systems, one tuned for code quality and the other for fast dynamic code generation.
- The linear scan algorithm for fast register allocation.
- An extensive set of 'C examples, which illustrate the utility of dynamic code generation and the ease of use of 'C in a variety of contexts.

- An analysis of the performance of tcc and tcc-generated dynamic code on several benchmarks. Measurements show that use of dynamic compilation can improve performance by almost an order of magnitude in some cases, and generally results in two- to four-fold speedups. The overhead of dynamic compilation is usually recovered in under 100 uses of the dynamic code; sometimes it can be recovered within one use.

2.3.2 Project history

The work on 'C was an outgrowth of earlier work done by Dawson Engler on VCODE [11] and DCG [14]. The paper on the language design for 'C appeared at POPL [12]. The first paper on the compiler appeared in 1997 at PLDI [28]. This pointed out that the performance of register allocation was important, which caused Max Poletto (one of the main designers of 'C) to come up with a new register allocation algorithm [30]. A journal paper documenting most aspects of the research appeared in 1999 [29]. The most complete document on 'C is Max Poletto's doctoral thesis [27], which was completed in June of 1999.

Tcc was released in stages. An early version first appeared in July 1996, and the first "real" version in December 1997. It developed a small community of active users. IBM used the linear scan algorithm in the Jikes/Jalapeno Java compiler.

2.4 Click

The Click project explores the construction of network routers using software running on standard PC hardware. This architecture makes sense for the following reasons. Edge routers are increasingly expected to perform a wide variety of complex tasks, including network address translation, encryption, filtering, acting as firewalls, and prioritizing traffic. Conventional routers built from special purpose hardware are not easily adaptable to these tasks. Even existing software routers, including those based on open-source operating systems, have relatively hard-wired packet forwarding paths that are difficult to extend, modify or configure.

Thus the Click project took on two challenges. First, to design a software structure that makes it easy to configure and control the packet forwarding tasks in a router; this structure should allow maximum flexibility and ease of programming. Second, to extract the maximum possible performance, despite running on non-specialized commodity PC hardware.

The project promised to have impact in a number of ways. First, it might ease the task of managing and programming routers. Second, it might decrease the cost of low- and medium-performance routers by use of commodity hardware. Third, it might form a flexible base for experimental networking research and development. The last point seemed the most important, since we had found the rigidity of existing router platforms raised the barriers to experimentation.

2.4.1 Click Architecture

Click routers are built from fine-grained components; this supports fine-grained extensions throughout the forwarding path. The components are packet processing modules called *elements*. The basic element interface is narrow, consisting mostly of functions for initialization and packet hand-off, but elements can extend it to support other functions (such as reporting queue lengths). To

build a router configuration, the user chooses a collection of elements and connects them into a directed graph. The graph's edges, which are called *connections*, represent possible paths for packet handoff. To extend a configuration, the user can write new elements or compose existing elements in new ways, much as UNIX allows one to build complex applications directly or by composing simpler ones using pipes.

Several aspects of the Click architecture were directly inspired by properties of routers. First, packet handoff along a connection may be initiated by either the source end (*push processing*) or the destination end (*pull processing*). This cleanly models most router packet flow patterns, and pull processing makes it possible to write composable packet schedulers. Second, the *flow-based router context* mechanism lets an element automatically locate other elements on which it depends; it is based on the observation that relevant elements are often connected by the flow of packets. These features make individual elements more powerful and configurations easier to write.

2.4.2 Project History

We started work on Click in February 1999. Our initial results, published at SOSP '99 [23] and in TOCS [21], highlighted the structure of a fully standards-compliant IP router, demonstrated the ease with which that configuration could be modified, and showed that it had performance far greater than existing software and hardware routers in a similar price range.

One of Click's strengths is its use of a declarative language to describe configurations. Early versions of Click simply parsed this language, directly producing C++ code to instantiate and connect elements. We later developed Click to take advantage of its language for three purposes [19]. First, to help generate efficient C++ code, since invariants are visible at the Click configuration level that are not visible to the C++ compiler. An example of this is that Click passes packets between elements using virtual function calls, which are expensive due to the extra level of indirection; it turns out that in any given configuration, the destinations of these calls are predictable and the level of indirection can be dispensed with. The second area of language-level optimization recognizes combinations of elements that can be replaced with more efficient specialized elements; this is a process much like peep-hole optimization. The final kind of language-level technique is checking configurations for correctness; for example, checking that a configuration maintains correct header field alignment as packets flow through the router. All of these techniques are implemented as pre-processing tools that analyze and potentially modify Click configurations. Eddie Kohler's Ph.D. thesis [18] describes these tools in more details and analyses their effect on performance.

Another area of Click development has been the construction of useful families of elements. These typically take the form of elements that each do a simple, well-defined task, and that are intended to be combined in open-ended ways. These families include elements for IPSec, IPv6, and TCP/IP header re-writing. The latter family, for example, uses a few basic elements to provide a wide array of functions: both port- and address-based Network Address Translation, transparent application-level proxies (such as transparent web caches), firewalls, and load balancing for server farms [22].

Click's use of a data-flow-like configuration language suggests that it should be possible to parallelize Click routers. Ideally, Click configurations written for uniprocessor routers could be automatically parallelized to take advantage of multiprocessors. This turns out to be possible. We

have extended Click to run on SMP PC hardware, and published results at USENIX 2001 [8]. Not only does multiprocessor Click automatically improve the performance of configurations written for uniprocessors without, it also makes it easy to express opportunities for increased parallelism by simple configuration modifications. For example, it is easy to express both partitioning unrelated packet flows among CPUs and pipelining single flows over a sequence of CPUs.

The Click project supports a complete distribution. The software is used by a number of projects within MIT as well as at many external research and commercial sites (see Section 3). After graduating with Ph.D.s, two of the students involved in the project started a company (Mazu Networks) that relies heavily on Click; the company's products analyze and control network traffic.

3 Technology transfer

This section summarizes the technology transfer through software distributions, personnel, and industrial relationships. The project histories in the previous sections already described the broader impact and technology transfer paths.

3.1 Software

- **Exokernel.** The first public version of the exokernel software was released in the beginning of 1998. Starting August 1 1998, we allowed researchers outside of MIT to contribute through anonymous CVS. The software was downloaded over the Web 1616 times since Aug. 1998, and another 1055 times through CVS. During most of 1998 and 1999, there were many contributions from inside and outside of MIT to the exokernel distributions. After the main students left (one to take a faculty position at Stanford, one to take a faculty position at CMU, and the remainder, except two, to found a company to commercialize the software), the intensity of software development dropped. Recently we have stopped active development and the maintenance of the exokernel distribution.

- **Prolac**. Since it was publicly released in 1999, there have been 271 distinct downloads of Prolac. The number of active users is small—a few small student projects. We have stopped the active maintenance of the Prolac distribution.

- **tcc**. The first real version was publicly released in December of 1997. There were 477 downloads from outside of MIT. Users were mostly academic: the biggest users included students of Peter Lee using it for CMU class projects, some people in the Tempo group at INRIA in France, 2 Ph.D. students in Denmark (Datalogisk Institut Kbenhavns Universitet and Aarhus), a Ph.D. student at University of Illinois at Urbana-Champaign, a master's student at U of Cincinnati, a student in Pune, India, who used it for a bachelor's project. The 'C mailing list had about 200 people on it. We stopped active maintenance of the 'C distribution in December 1999.

- **Click**. The first public release was 20 October 1999. There have been 5400 downloads and there are a number of active contributors outside of MIT to the Click distribution (through anonymous CVS). Users include Mazu Networks (as the platform for their commercial products), the ACIRI/ICSI Xorp project (as the forwarding engine for an open-source router), the

Flux group at the University of Utah (for use on a networking testbed), Knit (also University of Utah), the PDOS group at MIT (for development of ad-hoc networking protocols), the University of Colorado (to combine Click with ns-2 to support wireless infrastructure), Universiteit Gent (Click as infrastructure for diffserv work), and Princeton University (which uses the Click network drivers). Based on feedback and questions, a number of companies appear to be using Click for undisclosed projects: Checkpoint, Agilent, Siemens, Lancope, Wind River, Philips Research, Sonicity, Compaq. The universities that use Click include RPI, Purdue, MIT, Berkeley,Dublin City University, Stanford, University of Manitoba, and University of Cambridge (UK).

3.2 Personnel

Much of the technology has been transferred through students who have worked on the projects or through researchers that visited MIT to collaborate on the technology supported by this grant.

As an example in the first category is Xok/Cheetah. Tom Pinckney, Rusty Hunt, Doug Wyatt, and Josh Cates founded Exotec (later renamed to vividon, www.vividon.com), which produces high-performance low-cost video servers based on the Xok/Cheetah software developed at MIT under this contract.

As another example in the first category is Click. Max Poletto and Eddie Kohler founded Mazu Networks (http:://www.mazunetworks.com), which bases all its products on Click. Eddie Kohler also introduced researchers at ACIRI/ICSI to Click, which are now using it for the Xorp project, a forwarding engine for open-source routers.

As examples in the second category,we have collaborated closely with Steve Schwab et al. at TIS/NAI. They used the Xok software to develop a high-performance, secure active-network operating system for the DARPA active networks program. In addition to providing support through email and phone conversations, we hosted Steve Schwab, Daniel Maskit, and Hrishikesh Dandekar for extended periods.

3.3 Relationships

As result of much commercial interest in the DARPA-sponsored technology, we have fostered relationships with many companies. Companies that participate through grants include: Philips, Nokia, NTT, HP, Delta, and Acer. We had numerous technical meetings with Apple, Novel, and IBM to see how they could adopt the technology developed under this grant.

4 Publications

Exokernel Publications

[1] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie.

Application performance and flexibility on exokernel systems. pages 52–65.

[2] Dawson R. Engler. *The exokernel operating system architecture.* Ph.D. thesis, Massachusetts Institute of Technology, October 1998.

[3] Douglass Karl Wyatt. Shared libraries in an exokernel operating system Master's thesis, Massachusetts Institute of Technology, September 1997.

[4] John Jannotti. Applying exokernel principles to conventional operating systems. Master's thesis, Massachusetts Institute of Technology, February 1998.

[5] George M. Candea. Flexible and efficient sharing of protected abstractions. Master's thesis, Massachusetts Institute of Technology, May 1998.

[6] Charles L. Coffing. An x86 Protected Mode Virtual Machine Monitor for the MIT Exokernel. Master's thesis, Massachusetts Institute of Technology, May 1999.

[7] Benjie Chen. Multiprocessing with the exokernel operating system. Master's thesis, Massachusetts Institute of Technology, 2000.

Prolac Publications

[8] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. pages 3–13.

[9] Eddie Kohler. Prolac: a language for protocol compilation. Master's thesis, Massachusetts Institute of Technology, September 1997.

[10] David Rogers Montgomery, Jr. A fast Prolac TCP for the real world. Master's thesis, Massachusetts Institute of Technology, May 1999.

'C Publications

[11] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: a system for fast, flexible, and high-level dynamic code generation. pages 109–121.

[12] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. 21(2):324–369, March 1999.

[13] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. 21(5):895–913, September 1999.

[14] Massimiliano Poletto. *Language and compiler support for dynamic code generation.* Ph.D. thesis, Massachusetts Institute of Technology, June 1999.

Click Publications

[15] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. pages 217–231.

[16] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. 18(3):263–297, August 2000.

[17] Eddie Kohler, Benjie Chen, M. Frans Kaashoek, Robert Morris, and Massimiliano Poletto. Programming language techniques for modular router configurations. Technical Report MIT-LCS-TR-812, MIT Laboratory for Computer Science, August 2000.

[18] Eddie Kohler, Robert Morris, and Massimiliano Poletto. Modular components for network address translation. Technical report, MIT LCS Click Project, December 2000. http://www.pdos.lcs.mit.edu/papers/click-rewriter/.

[19] Eddie Kohler. *The Click modular router*. Ph. D. thesis, Massachusetts Institute of Technology, November 2000.

References

[1] ABBOTT, M. B., AND PETERSON, L. L. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking 1*, 1 (Feb. 1993), 4–19.

[2] ANDERSON, T. The case for application-specific operating systems. In *Third Workshop on Workstation Operating Systems* (1992), pp. 92–94.

[3] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO) (New York, Dec. 1995), ACM, pp. 267–284.

[4] BOLOGNESI, T., AND BRINKSMA, E. Introduction to the ISO specification language LOTOS. In *The formal description technique LOTOS*, P. H. J. van Eijk, C. A. Vissers, and M. Diaz, Eds. North-Holland, 1989, pp. 23–73.

[5] BRAKMO, L. S., O'MALLEY, S. W., AND PETERSON, L. L. TCP Vegas: new techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM 1994 Conference* (Aug. 1994), pp. 24–35.

[6] CASTELLUCCIA, C., DABBOUS, W., AND O'MALLEY, S. Generating efficient protocol code from an abstract specification. In *Proceedings of the ACM SIGCOMM 1996 Conference* (Aug. 1996), pp. 60–71.

[7] CHANKHUNTHOD, A., DANZIG, P., NEERDAELS, C., SCHWARTZ, M., AND WORRELL, K. A hierarchical Internet object cache. In *Proceedings of 1996 USENIX Technical Conference* (Jan. 1996), pp. 153–163.

[8] CHEN, B., AND MORRIS, R. Flexible control of parallelism in a multiprocessor pc router. pp. 333–346.

[9] CHERITON, D., AND DUDA, K. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation* (Nov. 1994), pp. 179–193.

[10] DEMBINSKI, P., AND BUDKOWSKI, S. Specification language Estelle. In *The formal description technique Estelle*, M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azema, and V. Chari, Eds. North-Holland, 1989, pp. 35–75.

[11] ENGLER, D. R. VCODE: a retargetable, extensible, very fast dynamic code generation system. pp. 160–170.

[12] ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 'C: A language for efficient, machine-independent dynamic code generation. pp. 131–144. An earlier version is available as MIT-LCS-TM-526.

[13] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE JR., J. Exokernel: An operating system architecture for application-specific resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO) (New York, Dec. 1995), ACM, pp. 251–266.

[14] ENGLER, D. R., AND PROEBSTING, T. A. DCG: an efficient, retargetable dynamic code generation system. pp. 263–272.

[15] HARTMAN, J., MONTZ, A., MOSBERGER, D., O'MALLEY, S., PETERSON, L., AND PROEBSTING, T. Scout: A communication-oriented operating system. Tech. Rep. TR 94-20, University of Arizona, Tucson, AZ, June 1994.

[16] HITZ, D. An NFS file server appliance. Tech. Rep. 3001, Network Applicance Corporation, Mar. 1995.

[17] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEÑO, H. M., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. pp. 52–65.

[18] KOHLER, E. *The Click modular router.* PhD thesis, Massachusetts Institute of Technology, Nov. 2000.

[19] KOHLER, E., CHEN, B., KAASHOEK, M. F., MORRIS, R., AND POLETTO, M. Programming language techniques for modular router configurations. Tech. Rep. MIT-LCS-TR-812, MIT Laboratory for Computer Science, Aug. 2000.

[20] KOHLER, E., KAASHOEK, M. F., AND MONTGOMERY, D. R. A readable TCP in the Prolac protocol language. pp. 3–13.

[21] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. 263–297.

[22] KOHLER, E., MORRIS, R., AND POLETTO, M. Modular components for network address translation. Tech. rep., MIT LCS Click Project, Dec. 2000. http://www.pdos.lcs.mit.edu/papers/click-rewriter/.

[23] MORRIS, R., KOHLER, E., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. pp. 217–231.

[24] MOSBERGER, D., PETERSON, L. L., BRIDGES, P. G., AND O'MALLEY, S. Analysis of techniques to improve protocol processing latency. In *Proceedings of the ACM SIGCOMM 1996 Conference* (Aug. 1996), pp. 73–84.

[25] MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (1996).

[26] PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. I/O-lite: a unified I/O buffering and caching system: Tech. Rep. `http://www.cs.rice.edu/~vivek/IO-lite.html`, Rice University, 1997.

[27] POLETTO, M. *Language and compiler support for dynamic code generation.* PhD thesis, Massachusetts Institute of Technology, June 1999.

[28] POLETTO, M., ENGLER, D. R., AND KAASHOEK, M. F. tcc: a system for fast, flexible, and high-level dynamic code generation. pp. 109–121.

[29] POLETTO, M., HSIEH, W. C., ENGLER, D. R., AND KAASHOEK, M. F. 'C and tcc: A language and compiler for dynamic code generation. 324–369.

[30] POLETTO, M., AND SARKAR, V. Linear scan register allocation. 895–913.

[31] POSTEL, J. Transmission Control Protocol: DARPA Internet Program protocol specification. RFC 793, IETF, Sept. 1981.

[32] SELTZER, M., ENDO, Y., SMALL, C., AND SMITH, K. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Oct. 1996), pp. 213–228.

[33] STEELE, JR., G. L. *Common LISP*, 2nd ed. Digital Press, Burlington, MA, 1990.

[34] STEVENS, W. R. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001, IETF, Jan. 1997.

[35] WRIGHT, G. R., AND STEVENS, W. R. *TCP/IP Illustrated, Volume 2: The Implementation.* Addison-Wesley, 1995.

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.